Practical Declarative Model Transformation With Tefkat

Michael Lawley[1] and Jim Steel[2]

[1] CRC for Enterprise Distributed Systems Technology (DSTC)**,
University of Queensland,
Brisbane, QLD 4072, Australia
michael@lawley.id.au
http://www.dstc.edu.au/

[2] INRIA/Irisa
University of Rennes 1, France
jsteel@irisa.fr

Abstract. We present Tefkat, an implementation of a language designed specifically for the transformation of MOF models using patterns and rules. The language adopts a declarative paradigm, wherein users may concern themselves solely with the relations between the models rather than needing to deal explicitly with issues such as order of rule execution and pattern searching/traversal of input models. In this paper, we demonstrate the language using a provided example and highlight a number of language features used in solving the problem, a simple object-to-relational mapping.

1 Introduction

Tefkat is the result of 5 years of research and development of languages for model transformation [1,2,3,4], most recently in the context of the OMG's QVT work [5]. In reaching the current point, one of the guiding principles has been that model transformation be treated as a specific problem, and that approaches treating it as a specific sub-problem of general-purpose programming will result in languages ill-suited for the specific issues that face model transformation.

Exploring different approaches to model transformations has revealed requirements, patterns and approaches in writing transformations that appeared very frequently when solving the prototypical examples of the problem space. The current approach attempts as much as possible to build these mechanisms into the language, in order that the programmer need not concern themselves with problems such as implementing algorithms for detecting input model patterns or ordering the application of their rules.

** The work reported in this paper has been funded in part by the Co-operative Research Centre for Enterprise Distributed Systems Technology (DSTC) through the Australian Federal Government's CRC Programme (Department of Education, Science, and Training).

In this paper we present a summary of the language and its features, using a mandatory example to illustrate how they combine to allow users to construct model transformations.

In Section 2 we present an overview of the language. In section 3 we elaborate on some of the details of the language and how they are used in solving the mandatory example. Section 4 presents a discussion of several aspects of Tefkat's implementation, including its concrete syntax and environment. The full text of the mandatory class-to-relational example may be found at the end of the paper following the conclusion.

2 Language Overview

The Tefkat language is declarative, logic-based, and defined in terms of a MOF metamodel. It has been specifically designed to address both the OMG's QVT RFP [6] and additional requirements identified as a result of a series of experiments with different transformation language approaches [1].

A Tefkat transformation specification effectively asserts a set of constraints that should hold over a collection of (disjoint) source and target extents (models). These constraints can:

- assert the existence of object instances in a target extent,
- assert the type of object instances in a target extent,
- assert the value(s) of object features,
- assert the relative order of values of an object's feature, and
- assert that a named relationship holds between one or more values (usually source and target object references).

A Tefkat language implementation uses these implied constraints to construct, if possible, a suitable set of target models that satisfy the constraints.

There are several aspects of the language worth noting:

- transformations do not specify a traversal order of the input models, nor an execution order for the rules – implementations must ensure that rules are executed in an order that satisfies the semantics,
- transformations are constructive – you cannot constrain an object to not exist, nor a feature to not have a particular value,
- it is not intended to describe or perform in-place model updates,
- change propagation can be supported through a model-merge process [7],
- the language is defined in terms of its abstract syntax (via a MOF metamodel). Thus, several concrete syntaxes are possible. This paper uses an SQL-inspired syntax.

Every transformation is expressed relative to three kinds of extents: one or more source extents, one or more target extents, and a single tracking extent. A transformation rule can query both source extents and the tracking extent, and can constrain/make assertions about both the target extents and the tracking

extent. Thus the tracking extent is special since it is the only extent that can be both queried and constrained.

More formally, a rule, r, can be considered to have two parts: the query, src, and the constraint, tgt, and two sets of variables: those that occur in the query, $\overline{x}$, and those that occur only in the constraint, $\overline{y}$. We can then write

$$r \equiv \forall \overline{x}\ src(\overline{x}) \rightarrow \exists \overline{y}\ tgt(\overline{x}, \overline{y})$$

3 Mandatory Example

In this section we introduce various aspects of the Tefkat language via fragments of the sample solution to the mandatory example. The full text of this solution can be found in Appendix A.

As shown in Figure 1, a transformation is a named entity with named parameters for the input and output models that participate in the transformation. Any number of metamodels may be imported by a transformation. This brings all the EClassifiers in these metamodels into consideration when class, datatype, and enum names are resolved.

```
TRANSFORMATION mtip05_class_to_relational: class -> relational

IMPORT http:///mtip05/class.ecore
IMPORT http:///mtip05/rdbms.ecore
```

Fig. 1. Transformation Definition

The transformation specification then contains any number of class definitions, rules, pattern definitions, and template definitions.

3.1 Class Definitions

Class definitions allow for the simple specification of ECore models and are part of the concrete syntax of Tefkat, but not part of its abstract syntax. Their main use is for definition of a transformation's tracking classes. Tracking classes are part of the mechanism used to represent the named relationships between source and target elements. Valid types for the features of these classes include all the types that are in-scope as a result of `IMPORT` statements plus the ECore datatypes corresponding to: `boolean`, `string`, `int`, `long`, `float`, and `double`.

Figure 2 shows the definition of two classes that are used for tracking relationships in the sample solution. We discuss these further in Section 3.4 below.

Note that for complex transformations one would normally create a separate meta-model defining these classes and import it into the transformation's namespace.

```
CLASS ClsToTbl {
  Class class;
  Table table;
};

CLASS AttrToCol {
  Class class;
  Attribute attr;
  Column col;
};
```

Fig. 2. (Tracking) Class Definitions

3.2 Rules

Rules are the primary action elements of the transformation. Broadly speaking, each rule consists of two constraints - source and target - that share variables. More specifically, the rule matches and then constrains a number of objects, either from the source model or from the trackings, and then *creates* (or ensures the existence of) a number of target model objects with a set of constraints.

```
RULE ClassAndTable(C, T)
  FORALL Class C {
           is_persistent: true;
           name: N;
         }
  MAKE Table T {
         name: N;
       }
  LINKING ClsToTbl WITH class = C, table = T;
```

Fig. 3. Rule Definition

For example, Figure 3 matches all instances of `Class` (in the default extent, `class`) for which the `is_persistent` attribute is true. and asserts that a `Table` with the same name must exist and that the `ClsToTbl` relationship holds for the corresponding `Class` and `Table` instances.

Since the semantics of rules requires the target to always hold whenever the source holds, we can use a target of `FALSE` to encode constraints that input models should satisfy in order for the transformation to be valid.

In the example, a non-persistent class with an association to itself would result in an infinite number of columns being created. Figure 4 shows a rule whose source pattern matches the illegal condition and whose target pattern is simply `FALSE`. Note the use of the built-in Pattern `println` to provide useful feedback in case the constraint is violated.

```
RULE constraint_no_reflexive_relations_on_non_persistent_classes
  FORALL Class C
  WHERE C.is_persistent = false
    AND ClassHasReference(C, C, _)
    AND println("Found a non-persistent class in relation (by association
or attribute) with itself: ", C)
  SET FALSE;
```

Fig. 4. Rule definition for a constraint

3.3 Pattern and Template Definitions

Pattern and template definitions are used to name and parameterise constraints that may be used in multiple rules. Pattern definitions correspond to source constraints and template definitions correspond to target constraints.

A pattern/template may be recursively defined. That is, it may directly or indirectly refer to itself. Such recursion is commonly used when matching recursive tree or graph structures like the `parent` reference of `Class` in the example.

Figure 5 illustrates several patterns used in the solution of the mandatory example. Note the recursive nature of the pattern `ClassHasAttr` to *drill down* into a `Class`'s attributes reflecting the recursive nature of the specification's rules 2, 4, and 5.

3.4 Trackings

Tracking classes are used to represent mapping relationships between source and target elements. While they may directly reflect a relationship established by a single rule (such as `ClassAndTable` in Figure 3), multiple rules may contribute to a single tracking relationship. This allows other rules that depend on that relationship to be decoupled from the details of how the relationship is established.

As discussed in [2,4], decoupling the rules that establish a mapping relationship from those that depend on that relationship is a key aspect of supporting maintainability and re-use of rules and transformations.

3.5 `FROM` clauses

For any non-trivial transformation one needs to be able to carefully control the number of objects that are created. In Tefkat this information is represented in the abstract syntax by an `Injection` term. The corresponding concrete syntax is the optional `FROM` clause. There will be exactly one object created for each unique tuple corresponding to a `FROM`.

In the case of `MAKE` clauses that do not contain explicit `FROM` clauses, an implicit `FROM` is constructed as follows: the label is the concatenation of the rule name and the name of the target instance variable, and the parameters are the

```
PATTERN ClassHasAttr(Class, Attr, Name, IsKey)
  WHERE ClassHasSimpleAttr(Class, Attr, Name, IsKey)
     OR ClassHasIncludedAttr(Class, Attr, Name, IsKey)
     OR ClassChildHasAttr(Class, Attr, Name, IsKey);

PATTERN ClassHasSimpleAttr(Class, Attr, Name, IsKey)
  FORALL Class Class {
           attrs: Attribute Attr {
             type: PrimitiveDataType _PT;
             name: Name;
             is_primary: IsKey;
           };
         };

PATTERN ClassHasIncludedAttr(Class, Attr, Name, IsKey)
  FORALL Class Class
  WHERE ClassHasReference(Class, Type, RefName)
    AND ClassHasAttr(Type, Attr, AttrName, IsKeyForType)
    AND IF Type.is_persistent = true
        THEN
          IsKeyForType = true AND
          IsKey = false
        ELSE
          IsKey = IsKeyForType
        ENDIF
    AND Name = join("_", RefName, AttrName);

PATTERN ClassChildHasAttr(Class, Attr, Name, IsKey)
  FORALL Class SubClass
  WHERE Class = SubClass.parent
    AND ClassHasAttr(SubClass, Attr, Name, IsKey)
    AND IsKey = false;
```

Fig. 5. Sample Pattern definitions

set of variables corresponding to source instances in the containing rule's FORALL clause.

For example, the implicit FROM clause for MAKE Table T in Figure 3 is: FROM ClassAndTable_T(C).

```
RULE MakeColumns
  WHERE ClassHasAttr(C, A, N, IsKey)
    AND ClsToTbl LINKS class = C, table = T
  MAKE Column Col FROM col(C, N) {
         name: N;
         type: A.type.name;
  }
  SET T.cols = Col,
      IF IsKey = true
      THEN
        SET T.pkey = Col
      ENDIF
  LINKING AttrToCol WITH class = C, attr = A, col = Col;
```

Fig. 6. Use of a FROM clause

Figure 6 shows a case where an explicit FROM is required. The originating Class and the path of attributes and associations, as encoded in the name bound to N uniquely identify the Columns to be created.

The use of an explicit FROM clause allows multiple rules to separately and independently assert the existence of a target object, with only a single object being actually created. Again, this enhances the maintainability and re-usablility of rules and transformations.

4 Language Implementation

4.1 Concrete Syntax

The concrete syntax of Tefkat was initially designed to feel familiar and comfortable to programmers with experience using SQL, another declarative language, and also to suggest an intuitive semantics that help direct the writing of rules.

The only major change to the syntax since its first specification has been the introduction of *object literals* as illustrated in Figure 7.

Object literals are pure syntactic sugar designed to make rules more succinct and readable since, with appropriate formatting, they expose explicit structure in the constraints being specified.

Figure 8 shows the equivalent constraint expressed without using object literal synyax.

Another concrete syntax feature that deserves special mention is the use of variables whose name begins with an underscore. These variables are termed

```
Class Class {
  attrs: Attribute Attr {
    type: PrimitiveDataType _PT;
    name: Name;
    is_primary: IsKey;
  };
}
```

Fig. 7. A simple *obect literal*

```
Class Class AND
Class.attrs = Attr AND
Attribute Attr AND
Attr.type = PT AND
PrimitiveDataType PT AND
Attr.name = Name AND
Attr.is_primary = IsKey
```

Fig. 8. An equivalent constraint for Figure 7

anonymous variables and references to them are, by definition, unique. That is, if the variable name `_PT`, for example, is used more than once in an individual rule, patterm, or temaplate, then each reference defines and refers to a different variable.

While not an error, the parser will emit a warning when an anonymous variable (except for the variables named by a single underscore) is used more than once. The parser will also emit a warning when a variable whose name does not begin with an underscore is used only once in a given rule. By naming variables to avoid these warnings, simple spelling mistakes and some copy-and-paste errors are more easily detected, which is a real bonus for a language that does not require variables to be explicitly declared.

4.2 Advanced Language Features

Tefkat is designed to support transformations that span meta-levels. There are two key features that enable this: reflection, and the *Any Type*.

Support for reflection comes in two parts. The simplest is allowing access to the reflective features that every object implicitly inherits from `EObject`. For example, you can access an object's container object with `O.eContainer()`, all its contained objects with `O.eContents()`, and its meta-class object with `O.eClass()`.

The more advanced aspect is the ability to use an arbitrary expression, prefixed by a dollar symbol, anywhere a type name or feature name may be used. In the simplest case, `O.$"name" = A` is equivalent to `O.name = N`. Other examples include `O.$join("_", N1, N2) = A` which gets the value of an attribute whose

name is the concatentation of `N1`, `"_"`, and `N2`, and `O1.name = N AND $N O2` which binds `O2` to all instances of the class named by the value of `N`.

The *Any Type* is represented by an underscore. It behaves like an implicit universal supertype of all types. This allows a transformation rule to match all objects regardless of their actual type and without requiring an explicit common supertype. To illustrate this, Figure 9 shows a transformation that makes a copy of an arbitrary input model.

```
TRANSFORMATION copy : src -> tgt

IMPORT http://www.eclipse.org/emf/2002/Ecore

CLASS ObjToObj {
  EObject src;
  EObject tgt;
};

RULE copyObjects
  FORALL _ Src
  MAKE   $Src.eClass() Tgt
  LINKING ObjToObj WITH src = Src, tgt = Tgt;

RULE copyAttributeValues
  WHERE ObjToObj LINKS src = Src, tgt = Tgt
    AND Src.eClass() = Class
    AND Class.eAllAttributes = Attr
    AND Attr.changeable = true
    AND Src.eIsSet(Attr) = true
    AND Value = Src.$Attr
  SET Tgt.$Attr = Value;

RULE copyObjectReferences
  WHERE ObjToObj LINKS src = Src, tgt = Tgt
    AND Src.eClass() = Class
    AND Class.eAllReferences = Ref
    AND Ref.changeable = true
    AND Src.eIsSet(Ref) = true
    AND Value = Src.$Ref
    AND ObjToObj LINKS src = Value, tgt = TgtValue
  SET Tgt.$Ref = TgtValue;
```

Fig. 9. A generic *copy* transformation

4.3 The Engine and Environment

The Tefkat engine is suitable for standalone use and is invokable from the command-line, but for most developers it will be used as part of a full-featured set of Eclipse plugins. These include a syntax-highlighting editor that is integrated with the parser to provide direct, linked feedback on parser errors and warnings. Figure 10 shows this editor, including a warning about a singleton variable use. Note the outline view to the right. Clicking on an entry in this view will cause the editor to jump to the appropriate line.

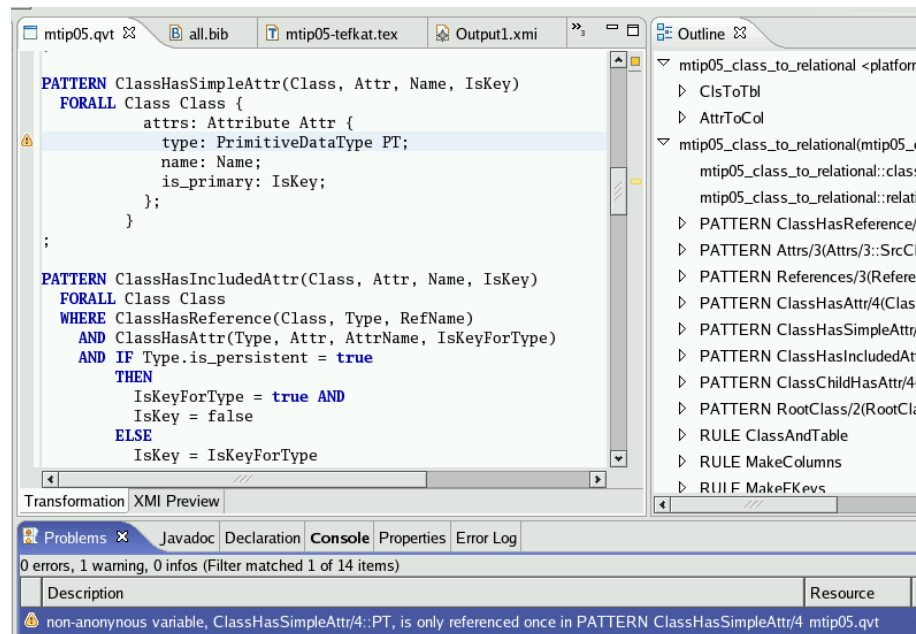

Fig. 10. The Tefkat editor for Eclipse

Also included with the Eclipse plugin is a source-level debugger, shown in Figure 11. Running the transformation in the debugger allows you to single step through the evaluation of each term in a rule. Variables and their bindings are shown in the view at the top right, while the stack displays the current term and the terms of the current rule that have been evaluated leading to this point.

While very useful, the debugger does suffer some limitations. Being a declarative logic-based language, the execution model is somewhat like that of Prolog. Thus the internal state is a set of trees rather than the stack of traditional procedural languages for which the Eclipse debugging framework is designed.

This, coupled with the need to re-order terms during evaluation for both efficiency and semantic correctness (for example, ensuring that a variable is bound to an object before attempting to get a feature's value), means that it

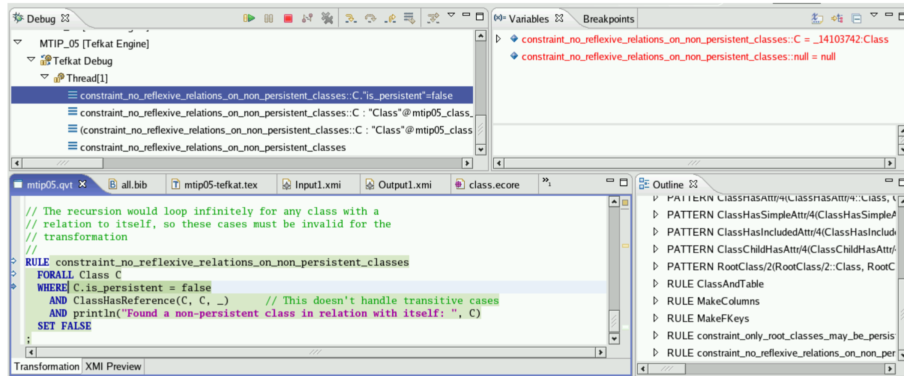

Fig. 11. The Tefkat source-level debugger for Eclipse

can sometimes be difficult to follow a rule's execution, although the integrated source highlighting helps a great deal.

To improve the debugging experience we would like to explore the use of annotations to describe the expected behaviour of parts of rules. This would be similar to the determinism declarations used in Mercury [8]. In the longer term, it may also be possible to adapt concepts from declarative debugging [9].

Tefkat is integrated with the Eclipse build system. Its configuration is stored as a model in the file `tefkat.xml`. This describes one or more transformation applications in terms of source and target models, an optional trace model for recording which rules and source elements were used to create which target elements, and any mappings required to translate URIs naming meta-models to resolvable URLs. The build integration allows a transformation to be re-run whenever the specification or any of the source models is updated.

Alternatively, the normal Eclipse launch mechanism can be used to manually execute a transformation. This is also how debugging mode is entered.

Finally, Tefkat includes several concessions to pragmatics. Firstly, as shown in Figure 4, Tefkat includes the pre-defined pattern `println` which always succeeds, binds no variables, and prints its arguments to the console. Its main use is as a *probe* for debugging.

Also useful for debugging is the ability to tell Tefkat to continue executing rather than aborting when a rule fails. This means that target and trace models are still generated and, although they result from buggy rules, they can be very useful for post-mortem debugging.

Another concession is the ability to invoke methods on objects, not just access features. This includes not just those methods defined in the meta-model, but also those that make up the Java implementation. Since Tefkat is built on EMF, this includes all the reflective methods from `EObject`. Note that calling methods that have side-effects is a dangerous and unpredictable thing to do since Tefkat makes no guarantees about evaluation order.

4.4 Limitations

One technical aspect of the language is the need for transformations to be stratified. Essentially this means that a rule (or pattern) cannot depend (directly or indirectly) on its own negation. For example, a rule cannot check that there are no instances of a tracking class, and then create an instance of that tracking class.

It is the need to be able to determine stratifiability of a transformation that gives rise to the limitations on querying elements in target extents. Tefkat's support for reflection means that determining *negative dependencies* in the face of arbitrary target extent queries would impose too great a cost. By limiting queries to source extents and the special tracking extent, this cost is avoided.

The price, however, is that complex transformations may need to store large amounts of information in the tracking classes. Future work will investigate whether it is practical to relax some of the limitations on querying target extents.

One possible way to mitigate this problem is to *stream* transformations. That is, instead of specifying a single large transformation that does everything, perform a series of smaller transformations. In this way, target extents from earlier transformations become queriable source extents in later transformations.

While this form of transformation composition can be done outside of the Tefkat language, we believe there are benefits to supporting it and other forms of composition directly in the language.

5 Conclusion

The example, typical of those used as both exemplary and motivating problems for model transformation, shows how the use of a declarative language allows transformation writers to focus their endeavours on the logic of the transformation rather than on how to facilitate its execution.

The example presented, although interesting, is by necessity small in scale. A number of other works are currently underway using the language and engine that are offering valuable feedback and serving to evaluate their ability to deal with large-scale examples. In [10], the authors have written Tefkat transformations to generate UML2 Testing Profile models from UML requirements and design models. In [7], the author addresses the problem of change propagation for Tefkat, and implements it using transformations themselves written in Tefkat. One transformation takes as input the updated source model, the original (possibly updated) target models and a newly generated target model, and the trace models for the original and new transformations and produces a delta model. The delta model represents the differences between the old and new target models. Based on heuristics and user feedback, a subsequent transformation produces a final target model that preserves any manual changes that may have been made between transformation executions.

In addition, the engine is also being used to manage the transformation between electronic health record formats and to generate Xforms for input of

health record information. The metamodels and transformations in each of these examples are both large and complex. The Xform transformation is also of particular interest because it takes two meta-models as input (a reference model, and an archetype model [11]), and produces an XML Schema-based model as output. Additionally, it needs to combine implicit hints from both input models to construct a useful ordering of input fields and labels in the resulting Xform.

Our experiences with Tefkat demonstrate that declarative transformation specification is both practical and productive. A declarative specification means you can concentrate on *what* the transformation should do rather than getting caught up in *how* the transformation should do it.

References

1. Gerber, A., Lawley, M., Raymond, K., Steel, J., Wood, A.: Transformation: The missing link of MDA. In Corradini, A., Ehrig, H., Kreowski, H.J., Rozemberg, G., eds.: Proc. 1st International Conference on Graph Transformation, ICGT'02. Volume 2505 of Lecture Notes in Computer Science., Springer Verlag (2002) 90–105
2. Duddy, K., Gerber, A., Lawley, M., Raymond, K., Steel, J.: Model transformation: A declarative, reusable patterns approach. In: Proc. 7th IEEE International Enterprise Distributed Object Computing Conference, EDOC 2003, Brisbane, Australia (2003) 174–195
3. Duddy, K., Gerber, A., Lawley, M., Raymond, K., Steel, J.: Declarative transformation for object-oriented models. In van Bommel, P., ed.: Transformation of Knowledge, Information, and Data: Theory and Applications. Idea Group Publishing (2004)
4. Lawley, M., Duddy, K., Gerber, A., Raymond, K.: Language features for re-use and maintainability of MDA transformations. In: OOPSLA workshop on Best Practices for Model-Driven Software Development, Vancouver, Canada (2004)
5. DSTC, IBM, CBOP: MOF Query/View/Transformation, initial submission (2003)
6. OMG: Request for Proposal: MOF 2.0 Query/Views/Transformations RFP. OMG Document: ad/02-04-10 (2002)
7. Metke, A.: Change propagation in the MDA: A model merging approach. Master's thesis, School of Information Technology and Electrical Engineering, The University of Queensland (2005)
8. Henderson, F., Somogyi, Z., Conway, T.: Determinism analysis in the Mercury compiler. In: Proceedings of the Australian Computer Science Conference, Melbourne, Australia (1996) 337–346
9. Naish, L.: A three-valued declarative debugging scheme. Technical Report 97/5, Department of Computer Science, University of Melbourne, Melbourne, Australia (1997)
10. Dai, Z.R.: Model-driven testing with UML 2.0. In Akehurst, D., ed.: Second European Workshop on Model Driven Architecture (MDA), Canterbury, Kent, University of Kent (2004) 179–187
11. Beale, T., Goodchild, A., Heard, S.: EHR design principles. http://titanium.dstc.edu.au/papers/ehr_design_principles.pdf (2002)

AppendixA

Complete Code for Class to RDBMS transformation

```
TRANSFORMATION mtip05_class_to_relational: class -> relational

IMPORT platform:/resource/mtip05/models/class.ecore
IMPORT platform:/resource/mtip05/models/rdbms.ecore

CLASS ClsToTbl {
  Class class;
  Table table;
};

CLASS AttrToCol {
  Class class;
  Attribute attr;
  Column col;
};

// ---------------------------------------------------------------------

// Class-typed Attributes and Associations are equivalent
//
PATTERN ClassHasReference(SrcClass, DstClass, RefName)
  WHERE Attrs(SrcClass, DstClass, RefName)
     OR References(SrcClass, DstClass, RefName)
;

PATTERN Attrs(SrcClass, DstClass, RefName)
  FORALL Class SrcClass { attrs: Attribute _ { type: Class DstClass; name: RefName; }; }
;

PATTERN References(SrcClass, DstClass, RefName)
  FORALL Association _ { src: SrcClass; dest: DstClass; name: RefName; }
;

// A Class "has" an Attribute for the purposes of the mapping if
//  1. It is directly owned and a primitive type
//  2. A referenced Class (via an Attribute or an Association) "has" the Attribute
//  3. The Class's children "have" the Attribute
//
PATTERN ClassHasAttr(Class, Attr, Name, IsKey)
  WHERE ClassHasSimpleAttr(Class, Attr, Name, IsKey)
     OR ClassHasIncludedAttr(Class, Attr, Name, IsKey)
     OR ClassChildHasAttr(Class, Attr, Name, IsKey)
;

PATTERN ClassHasSimpleAttr(Class, Attr, Name, IsKey)
```

```
  FORALL Class Class {
           attrs: Attribute Attr {
             type: PrimitiveDataType _PT;
             name: Name;
           };
         }
  WHERE IsKey = Attr.is_primary
;

PATTERN ClassHasIncludedAttr(Class, Attr, Name, IsKey)
  FORALL Class Class
  WHERE ClassHasReference(Class, Type, RefName)
    AND ClassHasAttr(Type, Attr, AttrName, IsKeyForType)
    AND IF Type.is_persistent = true
        THEN
          IsKeyForType = true AND
          IsKey = false
        ELSE
          IsKey = IsKeyForType
        ENDIF
    AND Name = join("_", RefName, AttrName)
;

PATTERN ClassChildHasAttr(Class, Attr, Name, IsKey)
  FORALL Class SubClass
  WHERE Class = SubClass.parent
    AND ClassHasAttr(SubClass, Attr, Name, IsKey)
    AND IsKey = false // Sub-classes cannot add keys
;


PATTERN RootClass(Class, Root)
  WHERE IF Parent = Class.parent
        THEN
          RootClass(Parent, Root)
        ELSE
          Root = Class
        ENDIF
;

// ----------------------------------------------------------------------

// 1. Classes that are marked as persistent in the source model
//    should be transformed into a single table of the same
//    name in the target model.
//
RULE ClassAndTable(C, T)
  FORALL Class C {
    is_persistent: true;
    name: N;
```

```
  }
  MAKE Table T {
    name: N;
  }
  LINKING ClsToTbl WITH class = C, table = T
;

// Transitively owned Attributes map to Columns
// If the Attribute is primary, so is the corresponding Column
// BUT ONLY IF THE Attribute's Class IS NOT PERSISTENT
//
RULE MakeColumns
  WHERE ClassHasAttr(C, A, N, IsKey)
    AND ClsToTbl LINKS class = C, table = T
  MAKE Column Col FROM col(C, N) {
         name: N;
         type: A.type.name;
  }
  SET T.cols = Col,
      IF IsKey = true
      THEN
        SET T.pkey = Col
      ENDIF
  LINKING AttrToCol WITH class = C, attr = A, col = Col
;

// References to persistent Classes result in an FKey to
// the corresponding Table
// Columns corresponding to primary Attributes of the target Class
// constitute the corresponding foreign key columns
//
RULE MakeFKeys
  WHERE ClassHasReference(Class, TgtClass, RefName)
    AND TgtClass.is_persistent = true
    AND RootClass(Class, SrcClass)
    AND ClsToTbl LINKS class = SrcClass, table = SrcTable
    AND ClsToTbl LINKS class = TgtClass, table = TgtTable
    // now determine the foreign key Columns
    AND AttrToCol LINKS class = SrcClass, attr = Attr, col = Col
    AND ClassHasAttr(TgtClass, Attr, _, true)
  MAKE FKey FKey FROM fkey(SrcTable, TgtTable, RefName) {
         references: TgtTable;
         cols: Col;
       }
       SET SrcTable.fkeys = FKey
;


// ----------------------------------------------------------------------
```

```
// Aborts the transformation in case the constraint is violated
//
RULE constraint_only_root_classes_may_be_persistent
  FORALL Class C
  WHERE C.is_persistent = true
    AND C.parent = _
    AND println("Found a persistent non-root class: ", C)
  SET FALSE
;

// The recursion would loop infinitely for any class with a
// relation to itself, so these cases must be invalid for the
// transformation
//
RULE constraint_no_reflexive_relations_on_non_persistent_classes
  FORALL Class C
  WHERE C.is_persistent = false
    AND ClassHasReference(C, C, _) // This doesn't handle transitive cases
    AND println("Found a non-persistent class in relation with itself: ", C)
  SET FALSE
;
```